CSE 431/531: Algorithm Analysis and Design (Spring 2022)

Divide-and-Conquer

Lecturer: Shi Li

Department of Computer Science and Engineering
University at Buffalo
Outline

1. Divide-and-Conquer
   - Counting Inversions
2. Quicksort and Selection
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
3. Polynomial Multiplication
4. Other Classic Algorithms using Divide-and-Conquer
5. Solving Recurrences
6. Computing $n$-th Fibonacci Number
<table>
<thead>
<tr>
<th>Greedy Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>mainly for combinatorial optimization problems</td>
</tr>
<tr>
<td>trivial algorithm runs in exponential time</td>
</tr>
<tr>
<td>greedy algorithm gives an efficient algorithm</td>
</tr>
<tr>
<td>main focus of analysis: correctness of algorithm</td>
</tr>
</tbody>
</table>
**Greedy Algorithm**
- mainly for combinatorial optimization problems
- trivial algorithm runs in exponential time
- greedy algorithm gives an efficient algorithm
- main focus of analysis: correctness of algorithm

**Divide-and-Conquer**
- not necessarily for combinatorial optimization problems
- trivial algorithm already runs in polynomial time
- divide-and-conquer gives a more efficient algorithm
- main focus of analysis: running time
Divide-and-Conquer

- **Divide**: Divide instance into many smaller instances
- **Conquer**: Solve each of smaller instances recursively and separately
- **Combine**: Combine solutions to small instances to obtain a solution for the original big instance
merge-sort($A, n$)

1: if $n = 1$ then
2: return $A$
3: else
4: $B \leftarrow$ merge-sort($A[1..\lfloor n/2 \rfloor], \lfloor n/2 \rfloor$)
5: $C \leftarrow$ merge-sort($A[\lceil n/2 \rceil + 1..n], \lceil n/2 \rceil$)
6: return merge($B, C, \lfloor n/2 \rfloor, \lfloor n/2 \rfloor$)

Divide: trivial
Conquer: 4, 5
Combine: 6
merge-sort($A, n$)

1: if $n = 1$ then
2: return $A$
3: else
4: $B \leftarrow$ merge-sort($A[1..\lfloor n/2 \rfloor], \lfloor n/2 \rfloor$)
5: $C \leftarrow$ merge-sort($A[\lfloor n/2 \rfloor + 1..n], \lceil n/2 \rceil$)
6: return merge($B, C, \lfloor n/2 \rfloor, \lceil n/2 \rceil$)

- Divide: trivial
- Conquer: 4, 5
- Combine: 6
Running Time for Merge-Sort

- Each level takes running time $O(n)$
- There are $O(\lg n)$ levels
- Running time $= O(n \lg n)$
- Better than insertion sort
Running Time for Merge-Sort Using Recurrence

- \( T(n) = \) running time for sorting \( n \) numbers, then

\[
T(n) = \begin{cases} 
O(1) & \text{if } n = 1 \\ 
T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n \geq 2 
\end{cases}
\]
**Running Time for Merge-Sort Using Recurrence**

- \( T(n) = \) running time for sorting \( n \) numbers, then
  
  \[
  T(n) = \begin{cases} 
    O(1) & \text{if } n = 1 \\
    T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n \geq 2
  \end{cases}
  \]

- With some tolerance of informality:
  
  \[
  T(n) = \begin{cases} 
    O(1) & \text{if } n = 1 \\
    2T(n/2) + O(n) & \text{if } n \geq 2
  \end{cases}
  \]
Running Time for Merge-Sort Using Recurrence

- \( T(n) = \text{running time for sorting } n \text{ numbers}, \) then

\[
T(n) = \begin{cases} 
O(1) & \text{if } n = 1 \\
T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n \geq 2 
\end{cases}
\]

- With some tolerance of informality:

\[
T(n) = \begin{cases} 
O(1) & \text{if } n = 1 \\
2T(n/2) + O(n) & \text{if } n \geq 2 
\end{cases}
\]

- Even simpler: \( T(n) = 2T(n/2) + O(n). \) (Implicit assumption: \( T(n) = O(1) \) if \( n \) is at most some constant.)
Running Time for Merge-Sort Using Recurrence

- \( T(n) \) = running time for sorting \( n \) numbers, then

\[
T(n) = \begin{cases} 
  O(1) & \text{if } n = 1 \\
  T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n \geq 2 
\end{cases}
\]

- With some tolerance of informality:

\[
T(n) = \begin{cases} 
  O(1) & \text{if } n = 1 \\
  2T(n/2) + O(n) & \text{if } n \geq 2 
\end{cases}
\]

- Even simpler: \( T(n) = 2T(n/2) + O(n) \). (Implicit assumption: \( T(n) = O(1) \) if \( n \) is at most some constant.)

- Solving this recurrence, we have \( T(n) = O(n \lg n) \) (we shall show how later)
Outline

1. Divide-and-Conquer
2. Counting Inversions
3. Quicksort and Selection
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
4. Polynomial Multiplication
5. Other Classic Algorithms using Divide-and-Conquer
6. Solving Recurrences
7. Computing $n$-th Fibonacci Number
Def. Given an array $A$ of $n$ integers, an inversion in $A$ is a pair $(i, j)$ of indices such that $i < j$ and $A[i] > A[j]$.
**Def.** Given an array $A$ of $n$ integers, an inversion in $A$ is a pair $(i, j)$ of indices such that $i < j$ and $A[i] > A[j]$.

**Counting Inversions**

**Input:** an sequence $A$ of $n$ numbers  
**Output:** number of inversions in $A$
Def. Given an array $A$ of $n$ integers, an inversion in $A$ is a pair $(i, j)$ of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

**Input:** an sequence $A$ of $n$ numbers

**Output:** number of inversions in $A$

Example:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8</td>
<td>15</td>
<td>9</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Def. Given an array $A$ of $n$ integers, an inversion in $A$ is a pair $(i, j)$ of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

- **Input:** an sequence $A$ of $n$ numbers
- **Output:** number of inversions in $A$

Example:

<table>
<thead>
<tr>
<th>10</th>
<th>8</th>
<th>15</th>
<th>9</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>15</td>
</tr>
</tbody>
</table>
Def. Given an array $A$ of $n$ integers, an inversion in $A$ is a pair $(i, j)$ of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

**Input:** an sequence $A$ of $n$ numbers

**Output:** number of inversions in $A$

Example:

10  8  15  9  12

8  9  10  12  15

4 inversions (for convenience, using numbers, not indices):

$(10, 8), (10, 9), (15, 9), (15, 12)$
Def. Given an array $A$ of $n$ integers, an inversion in $A$ is a pair $(i, j)$ of indices such that $i < j$ and $A[i] > A[j]$.

Counting Inversions

**Input:** an sequence $A$ of $n$ numbers

**Output:** number of inversions in $A$

Example:

4 inversions (for convenience, using numbers, not indices):

(10, 8), (10, 9), (15, 9), (15, 12)
Naive Algorithm for Counting Inversions

count-inversions\((A, n)\)

1: \(c \leftarrow 0\)
2: \textbf{for} every \(i \leftarrow 1\) to \(n - 1\) \textbf{do}
3: \hspace{1em} \textbf{for} every \(j \leftarrow i + 1\) to \(n\) \textbf{do}
4: \hspace{2em} \textbf{if} \(A[i] > A[j]\) \textbf{then} \(c \leftarrow c + 1\)
5: \textbf{return} \(c\)
Divide-and-Conquer

\[ p = \left\lfloor \frac{n}{2} \right\rfloor, B = A[1..p], C = A[p+1..n] \]

\[ \#\text{invs}(A) = \#\text{invs}(B) + \#\text{invs}(C) + m \]

\[ m = \left| \left\{ (i, j) : B[i] > C[j] \right\} \right| \]

**Q:** How fast can we compute \( m \), via trivial algorithm?

**A:** \( O(n^2) \)

- Can not improve the \( O(n^2) \) time for counting inversions.
Divide-and-Conquer

\[ p = \lfloor n/2 \rfloor, B = A[1..p], C = A[p + 1..n] \]

\[ \#\text{invs}(A) = \#\text{invs}(B) + \#\text{invs}(C) + m \]

\[ m = |\{(i, j) : B[i] > C[j]\}| \]

**Lemma** If both \( B \) and \( C \) are sorted, then we can compute \( m \) in \( O(n) \) time!
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$$B: \begin{array}{c} 3 \ 8 \ 12 \ 20 \ 32 \ 48 \end{array} \quad \text{total} = 0$$

$$C: \begin{array}{c} 5 \ 7 \ 9 \ 25 \ 29 \end{array}$$
Counting Inversions between \( B \) and \( C \)

Count pairs \( i, j \) such that \( B[i] > C[j] \):

\[
\begin{array}{cccccc}
B: & 3 & 8 & 12 & 20 & 32 & 48 \\
\end{array}
\quad \text{total} = 0
\]

\[
\begin{array}{cccccc}
C: & 5 & 7 & 9 & 25 & 29 \\
\end{array}
\]
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: 3 8 12 20 32 48

$C$: 5 7 9 25 29

$\text{total} = 0$

+0

3
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: 3 8 12 20 32 48

$C$: 5 7 9 25 29

total = 0

+0

3
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: \begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
\end{array}

$C$: \begin{array}{cccccc}
5 & 7 & 9 & 25 & 29 \\
\end{array}

$\text{total} = 0$

+0

\begin{array}{cc}
3 & 5 \\
\end{array}
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$: 

$\begin{align*}
B: & \quad 3 \quad 8 \quad 12 \quad 20 \quad 32 \quad 48 \\
C: & \quad 5 \quad 7 \quad 9 \quad 25 \quad 29 \\
\text{total} &= \ 0 \\
\end{align*}$

\[
\begin{array}{c|c|c|c|c|c|c}
& 3 & 8 & 12 & 20 & 32 & 48 \\
\hline
3 & 8 & 12 & 20 & 32 & 48 & +0 \\
5 & 7 & 9 & 25 & 29 & & \\
\hline
\end{array}
\]
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: 3 8 12 20 32 48

$C$: 5 7 9 25 29

total = 0

$+0$

3 5 7
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: 3 8 12 20 32 48  

$C$: 5 7 9 25 29  

$\text{total} = 0$  

$+0$  

3 5 7
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: 3 8 12 20 32 48  

$C$: 5 7 9 25 29  

+0 +2

$3 5 7 8$

$\text{total} = 2$
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$$B: \begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
\end{array}$$

$$C: \begin{array}{cccccc}
5 & 7 & 9 & 25 & 29 \\
\end{array}$$

Total = 2

+0 +2

\begin{array}{cccc}
3 & 5 & 7 & 8 \\
\end{array}$$
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: $\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
\end{array}$

$C$: $\begin{array}{cccccc}
5 & 7 & 9 & 25 & 29 \\
\end{array}$

$\text{total} = 2$

$+0 +2$

$\begin{array}{cccccc}
3 & 5 & 7 & 8 & 9 \\
\end{array}$
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

\[
\begin{array}{cccccc}
B: & 3 & 8 & 12 & 20 & 32 & 48 \\
C: & 5 & 7 & 9 & 25 & 29 \\
\end{array}
\]

\[\text{total} = 2\]
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: \[ \begin{array}{cccccc} 3 & 8 & 12 & 20 & 32 & 48 \end{array} \]

$C$: \[ \begin{array}{cccccc} 5 & 7 & 9 & 25 & 29 \end{array} \]

$+0 \quad +2 \quad +3$

$\text{total}= 5$
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: \[\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
\end{array}\]  total = 5

$C$: \[\begin{array}{cccccc}
5 & 7 & 9 & 25 & 29 \\
\end{array}\]

\[\begin{array}{cccccc}
+0 & +2 & +3 \\
\end{array}\]

\[\begin{array}{cccccc}
3 & 5 & 7 & 8 & 9 & 12 \\
\end{array}\]
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$: 

$B$: \[
\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
\end{array}
\]

$C$: \[
\begin{array}{cccccc}
5 & 7 & 9 & 25 & 29 \\
\end{array}
\]

Total = 8

\[
\begin{array}{cccccc}
3 & 5 & 7 & 8 & 9 & 12 & 20 \\
+0 & +2 & +3 & +3 \\
\end{array}
\]
Counting Inversions between \( B \) and \( C \)

Count pairs \( i, j \) such that \( B[i] > C[j] \):

\[
\begin{align*}
B: & \quad 3 \quad 8 \quad 12 \quad 20 \quad 32 \quad 48 \\
C: & \quad 5 \quad 7 \quad 9 \quad 25 \quad 29 \\
\end{align*}
\]

\[
\begin{align*}
& +0 \\
& +2 \\
& +3 \\
& +3 \\
\end{align*}
\]

\[
\begin{align*}
& 3 \quad 5 \quad 7 \quad 8 \quad 9 \quad 12 \quad 20 \\
\end{align*}
\]

\( \text{total} = 8 \)
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: 3 8 12 20 32 48  
$C$: 5 7 9 25 29

Total = 8

+0  +2  +3  +3

3 5 7 8 9 12 20 25
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: 3 8 12 20 32 48  
C: 5 7 9 25 29  

Total = 8

3 8 12 20 32 48
5 7 9 25 29
3 5 7 8 9 12 20 25
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$:  
3  8  12  20  32  48

$C$:  
5  7  9  25  29

$+0$  $+2$  $+3$  $+3$

$3  5  7  8  9  12  20  25  29$

total = 8
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$: 

$B$: \[ \begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
\end{array} \]

$C$: \[ \begin{array}{cccccc}
5 & 7 & 9 & 25 & 29 \\
\end{array} \]

$B$: $C$: total = 8

+0 +2 +3 +3
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$: $\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
\end{array}$

$\begin{array}{cccccc}
5 & 7 & 9 & 25 & 29 & \\
\end{array}$

$\begin{array}{cccccc}
+0 & +2 & +3 & +3 & +5 \\
\end{array}$

$\begin{array}{cccccccc}
3 & 5 & 7 & 8 & 9 & 12 & 20 & 25 & 29 & 32 \\
\end{array}$

$\text{total} = 13$
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

$B$: \[
\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
\end{array}
\]

$C$: \[
\begin{array}{cccccc}
5 & 7 & 9 & 25 & 29 & \\
\end{array}
\]

\[
\begin{array}{ccccccc}
\text{total} = 13 \\
+0 & +2 & +3 & +3 & +5 \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
3 & 5 & 7 & 8 & 9 & 12 & 20 & 25 & 29 & 32 \\
\end{array}
\]
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$:

\[
\begin{align*}
B: & \quad 3 & 8 & 12 & 20 & 32 & 48 \\
C: & \quad 5 & 7 & 9 & 25 & 29 \\
\end{align*}
\]

\[\text{total} = 18\]
Counting Inversions between $B$ and $C$

Count pairs $i, j$ such that $B[i] > C[j]$: 

$B$: 3 8 12 20 32 48

$C$: 5 7 9 25 29

$\text{total} = 18$

+0  +2  +3  +3  +5  +5

3  5  7  8  9  12  20  25  29  32  48
Count Inversions between $B$ and $C$

Procedure that merges $B$ and $C$ and counts inversions between $B$ and $C$ at the same time

merge-and-count($B, C, n_1, n_2$)

1: \( \text{count} \leftarrow 0; \)
2: \( A \leftarrow \text{array of size } n_1 + n_2; i \leftarrow 1; j \leftarrow 1 \)
3: \( \textbf{while } i \leq n_1 \text{ or } j \leq n_2 \text{ do} \)
4: \( \textbf{if } j > n_2 \text{ or } (i \leq n_1 \text{ and } B[i] \leq C[j]) \text{ then} \)
5: \( A[i + j − 1] \leftarrow B[i]; i \leftarrow i + 1 \)
6: \( \text{count} \leftarrow \text{count} + (j − 1) \)
7: \( \textbf{else} \)
8: \( A[i + j − 1] \leftarrow C[j]; j \leftarrow j + 1 \)
9: \( \textbf{return } (A, \text{count}) \)
A procedure that returns the sorted array of $A$ and counts the number of inversions in $A$:

\[
\text{sort-and-count}(A, n)
\]

1. \textbf{if} $n = 1$ \textbf{then}
2. \quad \textbf{return} $(A, 0)$
3. \textbf{else}
4. \quad $(B, m_1) \leftarrow \text{sort-and-count}(A[1..\lceil n/2 \rceil], \lceil n/2 \rceil)$
5. \quad $(C, m_2) \leftarrow \text{sort-and-count}(A[\lfloor n/2 \rfloor + 1..n], \lfloor n/2 \rfloor)$
6. \quad $(A, m_3) \leftarrow \text{merge-and-count}(B, C, \lfloor n/2 \rfloor, \lceil n/2 \rceil)$
7. \quad \textbf{return} $(A, m_1 + m_2 + m_3)$
A procedure that returns the sorted array of $A$ and counts the number of inversions in $A$:

**sort-and-count($A, n$)**

1. **if** $n = 1$ **then**
2. **return** $(A, 0)$
3. **else**
4. $(B, m_1) \leftarrow \text{sort-and-count}(A[1..\lfloor n/2 \rfloor], \lfloor n/2 \rfloor)$
5. $(C, m_2) \leftarrow \text{sort-and-count}(A[\lceil n/2 \rceil + 1..n], \lceil n/2 \rceil)$
6. $(A, m_3) \leftarrow \text{merge-and-count}(B, C, \lfloor n/2 \rfloor, \lceil n/2 \rceil)$
7. **return** $(A, m_1 + m_2 + m_3)$

- Divide: trivial
- Conquer: 4, 5
- Combine: 6, 7
sort-and-count($A, n$)

1: if $n = 1$ then
2: return ($A, 0$)
3: else
4: ($B, m_1$) ← sort-and-count($A[1..\lceil n/2 \rceil], \lfloor n/2 \rfloor$)
5: ($C, m_2$) ← sort-and-count($A[\lfloor n/2 \rfloor + 1..n], \lceil n/2 \rceil$)
6: ($A, m_3$) ← merge-and-count($B, C, \lfloor n/2 \rfloor, \lceil n/2 \rceil$)
7: return ($A, m_1 + m_2 + m_3$)

- Recurrence for the running time: $T(n) = 2T(n/2) + O(n)$
**sort-and-count**($A, n$)

1. **if** $n = 1$ **then**
2. **return** ($A, 0$)
3. **else**
4. ($B, m_1$) ← sort-and-count($A[1..\lfloor n/2 \rfloor], \lfloor n/2 \rfloor$)
5. ($C, m_2$) ← sort-and-count($A[\lfloor n/2 \rfloor + 1..n], \lceil n/2 \rceil$)
6. ($A, m_3$) ← merge-and-count($B, C, \lfloor n/2 \rfloor, \lceil n/2 \rceil$)
7. **return** ($A, m_1 + m_2 + m_3$)

- Recurrence for the running time: $T(n) = 2T(n/2) + O(n)$
- Running time = $O(n \lg n)$
Outline

1. Divide-and-Conquer
2. Counting Inversions
3. Quicksort and Selection
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
4. Polynomial Multiplication
5. Other Classic Algorithms using Divide-and-Conquer
6. Solving Recurrences
7. Computing $n$-th Fibonacci Number
Outline

1. Divide-and-Conquer
2. Counting Inversions
3. Quicksort and Selection
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
4. Polynomial Multiplication
5. Other Classic Algorithms using Divide-and-Conquer
6. Solving Recurrences
7. Computing $n$-th Fibonacci Number
# Quicksort vs Merge-Sort

<table>
<thead>
<tr>
<th>Divide</th>
<th>Merge Sort</th>
<th>Quicksort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conquer</td>
<td>Trivial</td>
<td>Separate small and big numbers</td>
</tr>
<tr>
<td>Combine</td>
<td>Recurse</td>
<td>Recurse</td>
</tr>
<tr>
<td></td>
<td>Merge 2 sorted arrays</td>
<td>Trivial</td>
</tr>
</tbody>
</table>
**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

| 29 | 82 | 75 | 64 | 38 | 45 | 94 | 69 | 25 | 76 | 15 | 92 | 37 | 17 | 85 |
**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

| 29 | 82 | 75 | 64 | 38 | 45 | 94 | 69 | 25 | 76 | 15 | 92 | 37 | 17 | 85 |
**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

<table>
<thead>
<tr>
<th>29</th>
<th>82</th>
<th>75</th>
<th><strong>64</strong></th>
<th>38</th>
<th>45</th>
<th>94</th>
<th>69</th>
<th>25</th>
<th>76</th>
<th>15</th>
<th>92</th>
<th>37</th>
<th>17</th>
<th>85</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>38</td>
<td>45</td>
<td>25</td>
<td>15</td>
<td>37</td>
<td>17</td>
<td><strong>64</strong></td>
<td>82</td>
<td>75</td>
<td>94</td>
<td>92</td>
<td>69</td>
<td>76</td>
<td>85</td>
</tr>
</tbody>
</table>
Quicksort Example

**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.
### Assumption

We can choose median of an array of size $n$ in $O(n)$ time.

<table>
<thead>
<tr>
<th>29</th>
<th>82</th>
<th>75</th>
<th>64</th>
<th>38</th>
<th>45</th>
<th>94</th>
<th>69</th>
<th>25</th>
<th>76</th>
<th>15</th>
<th>92</th>
<th>37</th>
<th>17</th>
<th>85</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>38</td>
<td>45</td>
<td>25</td>
<td>15</td>
<td>37</td>
<td>17</td>
<td>64</td>
<td>82</td>
<td>75</td>
<td>94</td>
<td>92</td>
<td>69</td>
<td>76</td>
<td>85</td>
</tr>
<tr>
<td>25</td>
<td>15</td>
<td>17</td>
<td>29</td>
<td>38</td>
<td>45</td>
<td>37</td>
<td>64</td>
<td>82</td>
<td>75</td>
<td>94</td>
<td>92</td>
<td>69</td>
<td>76</td>
<td>85</td>
</tr>
</tbody>
</table>
Quicksort

**quicksort**(*A, n*)

1: if *n* ≤ 1 then return *A*
2: *x* ← lower median of *A*
3: *A_L* ← array of elements in *A* that are less than *x* \ Divide
4: *A_R* ← array of elements in *A* that are greater than *x* \ Divide
5: *B_L* ← quicksort(*A_L*, length of *A_L*) \ Conquer
6: *B_R* ← quicksort(*A_R*, length of *A_R*) \ Conquer
7: *t* ← number of times *x* appear in *A*
8: return concatenation of *B_L*, *t* copies of *x*, and *B_R*
Quicksort

**quicksort**(A, n)

1: if \( n \leq 1 \) then return A
2: \( x \leftarrow \) lower median of A
3: \( A_L \leftarrow \) array of elements in A that are less than \( x \)  \( \text{// Divide} \)
4: \( A_R \leftarrow \) array of elements in A that are greater than \( x \)  \( \text{// Divide} \)
5: \( B_L \leftarrow \) quicksort\((A_L, \text{length of } A_L)\)  \( \text{// Conquer} \)
6: \( B_R \leftarrow \) quicksort\((A_R, \text{length of } A_R)\)  \( \text{// Conquer} \)
7: \( t \leftarrow \) number of times \( x \) appear in A
8: return concatenation of \( B_L \), \( t \) copies of \( x \), and \( B_R \)

- Recurrence \( T(n) \leq 2T(n/2) + O(n) \)
Quicksort

quicksort\((A, n)\)

1. if \(n \leq 1\) then return \(A\)
2. \(x \leftarrow\) lower median of \(A\)
3. \(A_L \leftarrow\) array of elements in \(A\) that are less than \(x\) \(\|\) Divide
4. \(A_R \leftarrow\) array of elements in \(A\) that are greater than \(x\) \(\|\) Divide
5. \(B_L \leftarrow\) quicksort\((A_L, \text{length of } A_L)\) \(\|\) Conquer
6. \(B_R \leftarrow\) quicksort\((A_R, \text{length of } A_R)\) \(\|\) Conquer
7. \(t \leftarrow\) number of times \(x\) appear \(A\)
8. return concatenation of \(B_L\), \(t\) copies of \(x\), and \(B_R\)

- Recurrence \(T(n) \leq 2T(n/2) + O(n)\)
- Running time = \(O(n \log n)\)
**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

**Q:** How to remove this assumption?
**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

**Q:** How to remove this assumption?

**A:**
1. There is an algorithm to find median in $O(n)$ time, using divide-and-conquer (we shall not talk about it; it is complicated and not practical)
**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

**Q:** How to remove this assumption?

**A:**
1. There is an algorithm to find median in $O(n)$ time, using divide-and-conquer (we shall not talk about it; it is complicated and not practical)
2. Choose a **pivot randomly** and pretend it is the median (it is practical)
quicksort($A, n$)

1: if $n \leq 1$ then return $A$
2: $x \leftarrow$ a random element of $A$ ($x$ is called a pivot)
3: $A_L \leftarrow$ array of elements in $A$ that are less than $x$ \ Divide
4: $A_R \leftarrow$ array of elements in $A$ that are greater than $x$ \ Divide
5: $B_L \leftarrow$ quicksort($A_L$, length of $A_L$) \ Conquer
6: $B_R \leftarrow$ quicksort($A_R$, length of $A_R$) \ Conquer
7: $t \leftarrow$ number of times $x$ appear $A$
8: return concatenation of $B_L$, $t$ copies of $x$, and $B_R$
Randomized Algorithm Model

**Assumption**  There is a procedure to produce a random real number in \([0, 1]\).

**Q:** Can computers really produce random numbers?
Randomized Algorithm Model

**Assumption**  There is a procedure to produce a random real number in \([0, 1]\).

**Q:** Can computers really produce random numbers?

**A:** No! The execution of a computer programs is deterministic!
Randomized Algorithm Model

**Assumption**  There is a procedure to produce a random real number in \([0, 1]\).

**Q:** Can computers really produce random numbers?

**A:** No! The execution of a computer programs is deterministic!

- In practice: use *pseudo-random-generator*, a deterministic algorithm returning numbers that “look like” random.
Randomized Algorithm Model

**Assumption** There is a procedure to produce a random real number in $[0, 1]$.

**Q:** Can computers really produce random numbers?

**A:** No! The execution of a computer programs is deterministic!

- In practice: use pseudo-random-generator, a deterministic algorithm returning numbers that “look like” random
- In theory: assume they can.
Quicksort Using A Random Pivot

quicksort\((A, n)\)

1: if \(n \leq 1\) then return \(A\)
2: \(x \leftarrow\) a random element of \(A\) (\(x\) is called a pivot)
3: \(A_L \leftarrow\) array of elements in \(A\) that are less than \(x\) \quad \| \quad \text{Divide}
4: \(A_R \leftarrow\) array of elements in \(A\) that are greater than \(x\) \quad \| \quad \text{Divide}
5: \(B_L \leftarrow\) quicksort\((A_L, \text{length of } A_L)\) \quad \| \quad \text{Conquer}
6: \(B_R \leftarrow\) quicksort\((A_R, \text{length of } A_R)\) \quad \| \quad \text{Conquer}
7: \(t \leftarrow\) number of times \(x\) appear \(A\)
8: return concatenation of \(B_L\), \(t\) copies of \(x\), and \(B_R\)

Lemma  \(\text{The expected running time of the algorithm is } O(n \lg n).\)
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

| 64 | 82 | 75 | 29 | 38 | 45 | 94 | 69 | 25 | 76 | 15 | 92 | 37 | 17 | 85 |
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

\[ i \quad j \]

\[
\begin{array}{cccccccccccccccc}
64 & 82 & 75 & 29 & 38 & 45 & 94 & 69 & 25 & 76 & 15 & 92 & 37 & 17 & 85 \\
\end{array}
\]
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- **In-Place Sorting Algorithm**: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

![Array elements](image)

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

```plaintext
i

17 37 64 29 38 45 94 69 25 76 15 92 75 82 85
```

To partition the array into two parts, we only need \( O(1) \) extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses "small" extra space.

```
17  37  15  29  38  45  94  69  25  76  64  92  75  82  85
```

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.

```
17 37 15 29 38 45 94 69 25 76 64 92 75 82 85
```
QuickSort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

```
17 37 15 29 38 45 64 69 25 76 94 92 75 82 85
```

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- **In-Place Sorting Algorithm**: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- **In-Place Sorting Algorithm**: an algorithm that only uses “small” extra space.

```
17 37 15 29 38 45 25 64 69 76 94 92 75 82 85
```

To partition the array into two parts, we only need $O(1)$ extra space.
Quicksort Can Be Implemented as an “In-Place” Sorting Algorithm

- In-Place Sorting Algorithm: an algorithm that only uses “small” extra space.

To partition the array into two parts, we only need $O(1)$ extra space.
\textbf{partition}(A, \ell, r)

1: \( p \leftarrow \) random integer between \( \ell \) and \( r \), swap \( A[p] \) and \( A[\ell] \)
2: \( i \leftarrow \ell, j \leftarrow r \)
3: \textbf{while true do}
4: \quad \textbf{while } i < j \text{ and } A[i] < A[j] \text{ do } j \leftarrow j - 1
5: \quad \textbf{if } i = j \text{ then break}
6: \quad \text{swap } A[i] \text{ and } A[j]; i \leftarrow i + 1
7: \quad \textbf{while } i < j \text{ and } A[i] < A[j] \text{ do } i \leftarrow i + 1
8: \quad \textbf{if } i = j \text{ then break}
9: \quad \text{swap } A[i] \text{ and } A[j]; j \leftarrow j - 1
10: \textbf{return } i
In-Place Implementation of Quick-Sort

\[ \text{quicksort}(A, \ell, r) \]

1. if \( \ell \geq r \) then return
2. \( m \leftarrow \text{partition}(A, \ell, r) \)
3. quicksort\((A, \ell, m - 1)\)
4. quicksort\((A, m + 1, r)\)

To sort an array \( A \) of size \( n \), call quicksort\((A, 1, n)\).

Note: We pass the array \( A \) by reference, instead of by copying.
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays
Merge-Sort is Not In-Place

To merge two arrays, we need a third array with size equaling the total size of two arrays

\[
\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
5 & 7 & 9 & 25 & 29 \\
\end{array}
\]
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays

```plaintext
<table>
<thead>
<tr>
<th>3</th>
<th>8</th>
<th>12</th>
<th>20</th>
<th>32</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7</td>
<td>9</td>
<td>25</td>
<td>29</td>
<td></td>
</tr>
</tbody>
</table>
```
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays

![Diagram](image)
Merge-Sort is Not In-Place

To merge two arrays, we need a third array with size equaling the total size of two arrays

```
3 8 12 20 32 48
5 7 9 25 29
3
```
To merge two arrays, we need a third array with size equaling the total size of two arrays.

```
3  8  12  20  32  48
5  7  9  25  29
3  5
```
Msort is Not In-Place

To merge two arrays, we need a third array with size equaling the total size of two arrays.

\[
\begin{array}{cccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
5 & 7 & 9 & 25 & 29 \\
3 & 5 \\
\end{array}
\]
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays.
Merge-Sort is Not In-Place

To merge two arrays, we need a third array with size equaling the total size of two arrays

```
3 8 12 20 32 48
5 7 9 25 29
3 5 7
```
Merge-Sort is Not In-Place

- To merge two arrays, we need a third array with size equaling the total size of two arrays.

```plaintext
3 8 12 20 32 48
5 7 9 25 29
3 5 7 8
```
Merge-Sort is Not In-Place

To merge two arrays, we need a third array with size equaling the total size of two arrays.

```
3  8  12  20  32  48
5  7  9  25  29
3  5  7  8
```
Merge-Sort is Not In-Place

To merge two arrays, we need a third array with size equaling the total size of two arrays.

\[
\begin{array}{cccccccc}
3 & 8 & 12 & 20 & 32 & 48 \\
5 & 7 & 9 & 25 & 29 \\
3 & 5 & 7 & 8 & 9 & 12 & 20 & 25 & 29
\end{array}
\]
Merge-Sort is Not In-Place

To merge two arrays, we need a third array with size equaling the total size of two arrays.
Outline

1. Divide-and-Conquer
2. Counting Inversions
3. Quicksort and Selection
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
4. Polynomial Multiplication
5. Other Classic Algorithms using Divide-and-Conquer
6. Solving Recurrences
7. Computing $n$-th Fibonacci Number
Comparison-Based Sorting Algorithms

Q: Can we do better than \( O(n \log n) \) for sorting?
Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.
Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

Comparison-Based Sorting Algorithms

- To sort, we are only allowed to compare two elements
- We cannot use “internal structures” of the elements
Lemma  The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$. 

Bob has one number $x$ in his hand, $x \in \{1, 2, 3, \ldots, N\}$. You can ask Bob “yes/no” questions about $x$.

Q: How many questions do you need to ask Bob in order to know $x$?

A: $\lceil \log_2 N \rceil$. 

$x = 1?$

$x \leq 2?$

$x = 3?$

1 2 3 4
Lemma  The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number $x$ in his hand, $x \in \{1, 2, 3, \ldots, N\}$. 

  Q: How many questions do you need to ask Bob in order to know $x$?

  A: $\lceil \log_2 N \rceil$. 

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1?</td>
<td>x ≤ 2?</td>
<td>x = 3?</td>
<td></td>
</tr>
</tbody>
</table>
Lemma The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \log n)$.

- Bob has one number $x$ in his hand, $x \in \{1, 2, 3, \ldots, N\}$.
- You can ask Bob “yes/no” questions about $x$.  

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 1$?</td>
<td>$x \leq 2$?</td>
</tr>
<tr>
<td>$x = 3$?</td>
<td>$1 \ 2 \ 3 \ 4$</td>
</tr>
</tbody>
</table>

Q: How many questions do you need to ask Bob in order to know $x$?  
A: $\lceil \log_2 N \rceil$.  

32/73
Lemma  The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number $x$ in his hand, $x \in \{1, 2, 3, \ldots, N\}$.
- You can ask Bob “yes/no” questions about $x$.

Q: How many questions do you need to ask Bob in order to know $x$?
Lemma  The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number $x$ in his hand, $x \in \{1, 2, 3, \ldots, N\}$.
- You can ask Bob “yes/no” questions about $x$.

Q: How many questions do you need to ask Bob in order to know $x$?

A: $\lceil \log_2 N \rceil$. 
**Lemma**  The (worst-case) running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$.

- Bob has one number $x$ in his hand, $x \in \{1, 2, 3, \cdots, N\}$.
- You can ask Bob “yes/no” questions about $x$.

**Q:** How many questions do you need to ask Bob in order to know $x$?

**A:** $\lceil \log_2 N \rceil$.
**Q:** Can we do better than $O(n \log n)$ for sorting?

**A:** No, for comparison-based sorting algorithms.

- Bob has a permutation $\pi$ over $\{1, 2, 3, \cdots, n\}$ in his hand.
- You can ask Bob “yes/no” questions about $\pi$. 
Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation $\pi$ over \{1, 2, 3, \ldots, n\} in his hand.
- You can ask Bob “yes/no” questions about $\pi$.

Q: How many questions do you need to ask in order to get the permutation $\pi$?
Comparison-Based Sorting Algorithms

**Q:** Can we do better than $O(n \log n)$ for sorting?

**A:** No, for comparison-based sorting algorithms.

- Bob has a permutation $\pi$ over $\{1, 2, 3, \ldots, n\}$ in his hand.
- You can ask Bob “yes/no” questions about $\pi$.

**Q:** How many questions do you need to ask in order to get the permutation $\pi$?

**A:** $\log_2 n! = \Theta(n \log n)$
Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation $\pi$ over $\{1, 2, 3, \ldots, n\}$ in his hand.
- You can ask Bob questions of the form “does $i$ appear before $j$ in $\pi$?”
Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation $\pi$ over $\{1, 2, 3, \ldots, n\}$ in his hand.
- You can ask Bob questions of the form "does $i$ appear before $j$ in $\pi$?"

Q: How many questions do you need to ask in order to get the permutation $\pi$?
Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

- Bob has a permutation $\pi$ over \( \{1, 2, 3, \cdots , n\} \) in his hand.
- You can ask Bob questions of the form “does $i$ appear before $j$ in $\pi$?”,

Q: How many questions do you need to ask in order to get the permutation $\pi$?

A: At least $\log_2 n! = \Theta(n \log n)$
Outline

1. Divide-and-Conquer
2. Counting Inversions
3. Quicksort and Selection
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
4. Polynomial Multiplication
5. Other Classic Algorithms using Divide-and-Conquer
6. Solving Recurrences
7. Computing $n$-th Fibonacci Number
Selection Problem

**Input:** a set $A$ of $n$ numbers, and $1 \leq i \leq n$

**Output:** the $i$-th smallest number in $A$

Sorting solves the problem in time $O(n \lg n)$. Our goal: $O(n)$ running time
Selection Problem

**Input:** a set $A$ of $n$ numbers, and $1 \leq i \leq n$

**Output:** the $i$-th smallest number in $A$

- Sorting solves the problem in time $O(n \lg n)$. 
**Selection Problem**

**Input:** a set $A$ of $n$ numbers, and $1 \leq i \leq n$

**Output:** the $i$-th smallest number in $A$

- Sorting solves the problem in time $O(n \lg n)$.
- Our goal: $O(n)$ running time
Recall: Quicksort with Median Finder

\[
\text{quicksort}(A, n)
\]

1. **if** \( n \leq 1 \) **then return** \( A \)
2. \( x \leftarrow \) lower median of \( A \)
3. \( A_L \leftarrow \) elements in \( A \) that are less than \( x \) ▶ Divide
4. \( A_R \leftarrow \) elements in \( A \) that are greater than \( x \) ▶ Divide
5. \( B_L \leftarrow \) quicksort\((A_L, A_L.\text{size})\) ▶ Conquer
6. \( B_R \leftarrow \) quicksort\((A_R, A_R.\text{size})\) ▶ Conquer
7. \( t \leftarrow \) number of times \( x \) appear \( A \)
8. **return** the array obtained by concatenating \( B_L \), the array containing \( t \) copies of \( x \), and \( B_R \)
Selection Algorithm with Median Finder

selection(A, n, i)

1: if n = 1 then return A
2: x ← lower median of A
3: A_L ← elements in A that are less than x ▶ Divide
4: A_R ← elements in A that are greater than x ▶ Divide
5: if i ≤ A_L.size then
6: return selection(A_L, A_L.size, i) ▶ Conquer
7: else if i > n − A_R.size then
8: return selection(A_R, A_R.size, i − (n − A_R.size)) ▶ Conquer
9: else
10: return x
Selection Algorithm with Median Finder

**selection**($A, n, i$)

1: if $n = 1$ then return $A$
2: $x \leftarrow$ lower median of $A$
3: $A_L \leftarrow$ elements in $A$ that are less than $x$ \hspace{1cm} ▶ Divide
4: $A_R \leftarrow$ elements in $A$ that are greater than $x$ \hspace{1cm} ▶ Divide
5: if $i \leq A_L$.size then
6: \hspace{1cm} return selection($A_L, A_L$.size, $i$) \hspace{1cm} ▶ Conquer
7: else if $i > n - A_R$.size then
8: \hspace{1cm} return selection($A_R, A_R$.size, $i - (n - A_R$.size)) \hspace{1cm} ▶ Conquer
9: else
10: \hspace{1cm} return $x$

- Recurrence for selection: $T(n) = T(n/2) + O(n)$
Selection Algorithm with Median Finder

\[ \text{selection}(A, n, i) \]

1: \textbf{if} \( n = 1 \) \textbf{then} \textbf{return} \( A \)
2: \( x \leftarrow \) lower median of \( A \)
3: \( A_L \leftarrow \) elements in \( A \) that are less than \( x \) \hfill \( \triangleright \) Divide
4: \( A_R \leftarrow \) elements in \( A \) that are greater than \( x \) \hfill \( \triangleright \) Divide
5: \textbf{if} \( i \leq A_L.\text{size} \) \textbf{then}
6: \textbf{return} \text{selection}(A_L, A_L.\text{size}, i) \hfill \( \triangleright \) Conquer
7: \textbf{else if} \( i > n - A_R.\text{size} \) \textbf{then}
8: \textbf{return} \text{selection}(A_R, A_R.\text{size}, i - (n - A_R.\text{size})) \hfill \( \triangleright \) Conquer
9: \textbf{else}
10: \textbf{return} \( x \)

- Recurrence for selection: \( T(n) = T(n/2) + O(n) \)
- Solving recurrence: \( T(n) = O(n) \)
Randomized Selection Algorithm

\textbf{selection}(A, n, i)

1: \textbf{if} $n = 1$ \textbf{then} return $A$
2: $x \leftarrow$ random element of $A$ (called pivot)
3: $A_L \leftarrow$ elements in $A$ that are less than $x$ \hspace{1cm} \triangleright \text{Divide}
4: $A_R \leftarrow$ elements in $A$ that are greater than $x$ \hspace{1cm} \triangleright \text{Divide}
5: \textbf{if} $i \leq A_L$.size \textbf{then}
6: \hspace{1cm} \textbf{return} selection($A_L$, $A_L$.size, $i$) \hspace{1cm} \triangleright \text{Conquer}
7: \textbf{else if} $i > n - A_R$.size \textbf{then}
8: \hspace{1cm} \textbf{return} selection($A_R$, $A_R$.size, $i - (n - A_R$.size)) \hspace{1cm} \triangleright \text{Conquer}
9: \textbf{else}
10: \hspace{1cm} \textbf{return} $x$

\text{expected running time} = \mathcal{O}(n)
Randomized Selection Algorithm

**selection**(*A*, *n*, *i*)

1: **if** *n* = 1 **then** return *A*
2: *x* ← random element of *A* (called pivot)
3: *A*<sub>L</sub> ← elements in *A* that are less than *x* ▶ Divide
4: *A*<sub>R</sub> ← elements in *A* that are greater than *x* ▶ Divide
5: **if** *i* ≤ *A*<sub>L</sub>.size **then**
7: **else if** *i* > *n* − *A*<sub>R</sub>.size **then**
8: return **selection**(*A*<sub>R</sub>, *A*<sub>R</sub>.size, *i* − (*n* − *A*<sub>R</sub>.size)) ▶ Conquer
9: **else**
10: return *x*

- expected running time = \(O(n)\)
Outline

1. Divide-and-Conquer
2. Counting Inversions
3. Quicksort and Selection
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
4. Polynomial Multiplication
5. Other Classic Algorithms using Divide-and-Conquer
6. Solving Recurrences
7. Computing $n$-th Fibonacci Number
Polynomial Multiplication

**Input:** two polynomials of degree \( n - 1 \)

**Output:** product of two polynomials
Polynomial Multiplication

Input: two polynomials of degree $n - 1$

Output: product of two polynomials

Example:

$$(3x^3 + 2x^2 - 5x + 4) \times (2x^3 - 3x^2 + 6x - 5)$$
Polynomial Multiplication

**Input:** two polynomials of degree $n - 1$

**Output:** product of two polynomials

**Example:**

$$(3x^3 + 2x^2 - 5x + 4) \times (2x^3 - 3x^2 + 6x - 5)$$

$$= 6x^6 - 9x^5 + 18x^4 - 15x^3$$
$$+ 4x^5 - 6x^4 + 12x^3 - 10x^2$$
$$- 10x^4 + 15x^3 - 30x^2 + 25x$$
$$+ 8x^3 - 12x^2 + 24x - 20$$

$$= 6x^6 - 5x^5 + 2x^4 + 20x^3 - 52x^2 + 49x - 20$$
Polynomial Multiplication

**Input:** two polynomials of degree $n - 1$

**Output:** product of two polynomials

**Example:**

$$(3x^3 + 2x^2 - 5x + 4) \times (2x^3 - 3x^2 + 6x - 5)$$

$$= 6x^6 - 9x^5 + 18x^4 - 15x^3 + 4x^5 - 6x^4 + 12x^3 - 10x^2 - 10x^4 + 15x^3 - 30x^2 + 25x + 8x^3 - 12x^2 + 24x - 20$$

$$= 6x^6 - 5x^5 + 2x^4 + 20x^3 - 52x^2 + 49x - 20$$

**Input:** $(4, -5, 2, 3), (-5, 6, -3, 2)$

**Output:** $(-20, 49, -52, 20, 2, -5, 6)$
Naïve Algorithm

\textbf{polynomial-multiplication}(A, B, n)

1: let $C[k] \leftarrow 0$ for every $k = 0, 1, 2, \cdots, 2n - 2$
2: \textbf{for} $i \leftarrow 0$ to $n - 1$ \textbf{do}
3: \hspace{1em} \textbf{for} $j \leftarrow 0$ to $n - 1$ \textbf{do}
4: \hspace{2em} $C[i + j] \leftarrow C[i + j] + A[i] \times B[j]$
5: \textbf{return} $C$

\text{Running time: } O(n^2)
Naïve Algorithm

polynomial-multiplication\((A, B, n)\)

1: let \(C[k] \leftarrow 0\) for every \(k = 0, 1, 2, \cdots, 2n - 2\)
2: for \(i \leftarrow 0\) to \(n - 1\) do
3: for \(j \leftarrow 0\) to \(n - 1\) do
4: \(C[i + j] \leftarrow C[i + j] + A[i] \times B[j]\)
5: return \(C\)

Running time: \(O(n^2)\)
Divide-and-Conquer for Polynomial Multiplication

\[ p(x) = 3x^3 + 2x^2 - 5x + 4 = (3x + 2)x^2 + (-5x + 4) \]
\[ q(x) = 2x^3 - 3x^2 + 6x - 5 = (2x - 3)x^2 + (6x - 5) \]
Divide-and-Conquer for Polynomial Multiplication

\[ p(x) = 3x^3 + 2x^2 - 5x + 4 = (3x + 2)x^2 + (-5x + 4) \]
\[ q(x) = 2x^3 - 3x^2 + 6x - 5 = (2x - 3)x^2 + (6x - 5) \]

- \( p(x) \): degree of \( n - 1 \) (assume \( n \) is even)
- \( p(x) = p_H(x)x^{n/2} + p_L(x), \)
- \( p_H(x), p_L(x) \): polynomials of degree \( n/2 - 1 \).
Divide-and-Conquer for Polynomial Multiplication

\[ p(x) = 3x^3 + 2x^2 - 5x + 4 = (3x + 2)x^2 + (-5x + 4) \]
\[ q(x) = 2x^3 - 3x^2 + 6x - 5 = (2x - 3)x^2 + (6x - 5) \]

- \( p(x) \): degree of \( n - 1 \) (assume \( n \) is even)
- \( p(x) = p_H(x)x^{n/2} + p_L(x) \),
- \( p_H(x), p_L(x) \): polynomials of degree \( n/2 - 1 \).

\[ pq = (p_H x^{n/2} + p_L)(q_H x^{n/2} + q_L) \]
Divide-and-Conquer for Polynomial Multiplication

\[
p(x) = 3x^3 + 2x^2 - 5x + 4 = (3x + 2)x^2 + (-5x + 4)
\]
\[
q(x) = 2x^3 - 3x^2 + 6x - 5 = (2x - 3)x^2 + (6x - 5)
\]

- \( p(x) \): degree of \( n - 1 \) (assume \( n \) is even)
- \( p(x) = p_H(x)x^{n/2} + p_L(x) \),
- \( p_H(x), p_L(x) \): polynomials of degree \( n/2 - 1 \).

\[
pq = (p_Hx^{n/2} + p_L)(q_Hx^{n/2} + q_L)
\]
\[
= p_Hq_Hx^n + (p_Hq_L + p_Lq_H)x^{n/2} + p_Lq_L
\]
Divide-and-Conquer for Polynomial Multiplication

\[ pq = (p_H x^{n/2} + p_L) (q_H x^{n/2} + q_L) = p_H q_H x^n + (p_H q_L + p_L q_H) x^{n/2} + p_L q_L \]
Divide-and-Conquer for Polynomial Multiplication

\[ pq = (p_H x^{n/2} + p_L) (q_H x^{n/2} + q_L) = p_H q_H x^n + (p_H q_L + p_L q_H) x^{n/2} + p_L q_L \]

\[ \text{multiply}(p, q) = \text{multiply}(p_H, q_H) \times x^n + (\text{multiply}(p_H, q_L) + \text{multiply}(p_L, q_H)) \times x^{n/2} + \text{multiply}(p_L, q_L) \]
Divide-and-Conquer for Polynomial Multiplication

\[ pq = (p_H x^{n/2} + p_L)(q_H x^{n/2} + q_L) \]
\[ = p_H q_H x^n + (p_H q_L + p_L q_H) x^{n/2} + p_L q_L \]

\[
\text{multiply}(p, q) = \text{multiply}(p_H, q_H) \times x^n \\
+ (\text{multiply}(p_H, q_L) + \text{multiply}(p_L, q_H)) \times x^{n/2} \\
+ \text{multiply}(p_L, q_L)
\]

\[ \text{Recurrence: } T(n) = 4T(n/2) + O(n) \]
Divide-and-Conquer for Polynomial Multiplication

\[
pq = (p_H x^{n/2} + p_L)(q_H x^{n/2} + q_L) = p_H q_H x^n + (p_H q_L + p_L q_H) x^{n/2} + p_L q_L
\]

\[
multiply(p, q) = multiply(p_H, q_H) \times x^n + (multiply(p_H, q_L) + multiply(p_L, q_H)) \times x^{n/2} + multiply(p_L, q_L)
\]

- Recurrence: \( T(n) = 4T(n/2) + O(n) \)
- \( T(n) = O(n^2) \)
Reduce Number from 4 to 3
Reduce Number from 4 to 3

\[ pq = \left( p_H x^{n/2} + p_L \right) \left( q_H x^{n/2} + q_L \right) \]

\[ = p_H q_H x^n + (p_H q_L + p_L q_H) x^{n/2} + p_L q_L \]
Reduce Number from 4 to 3

\[ pq = (p_H x^{n/2} + p_L) (q_H x^{n/2} + q_L) \]
\[ = p_H q_H x^n + (p_H q_L + p_L q_H) x^{n/2} + p_L q_L \]

\[ p_H q_L + p_L q_H = (p_H + p_L)(q_H + q_L) - p_H q_H - p_L q_L \]
Divide-and-Conquer for Polynomial Multiplication

\[ r_H = \text{multiply}(p_H, q_H) \]
\[ r_L = \text{multiply}(p_L, q_L) \]
\[ \text{multiply}(p, q) = r_H \times x^{n} + \left( \text{multiply}(p_H + p_L, q_H + q_L) - r_H - r_L \right) \times x^{n/2} + r_L \]

Solving Recurrence:
\[ T(n) = 3T(n/2) + O(n) \]
\[ T(n) = O(n \log_3 2) = O(n^{1.585}) \]
Divide-and-Conquer for Polynomial Multiplication

\[
r_H = \text{multiply}(p_H, q_H) \\
r_L = \text{multiply}(p_L, q_L)
\]
Divide-and-Conquer for Polynomial Multiplication

\[ r_H = \text{multiply}(p_H, q_H) \]
\[ r_L = \text{multiply}(p_L, q_L) \]

\[
multiply(p, q) = r_H \times x^n + (\text{multiply}(p_H + p_L, q_H + q_L) - r_H - r_L) \times x^{n/2} + r_L
\]
Divide-and-Conquer for Polynomial Multiplication

\[ r_H = \text{multiply}(p_H, q_H) \]
\[ r_L = \text{multiply}(p_L, q_L) \]

\[
\text{multiply}(p, q) = r_H \times x^n \\
+ (\text{multiply}(p_H + p_L, q_H + q_L) - r_H - r_L) \times x^{n/2} \\
+ r_L
\]

- Solving Recurrence: \( T(n) = 3T(n/2) + O(n) \)
Divide-and-Conquer for Polynomial Multiplication

\[ r_H = \text{multiply}(p_H, q_H) \]
\[ r_L = \text{multiply}(p_L, q_L) \]

\[
multiply(p, q) = r_H \times x^n + \left( \text{multiply}(p_H + p_L, q_H + q_L) - r_H - r_L \right) \times x^{n/2} + r_L
\]

- Solving Recurrence: \( T(n) = 3T(n/2) + O(n) \)
- \( T(n) = O(n^{\log_2 3}) = O(n^{1.585}) \)
Assumption \( n \) is a power of 2. Arrays are 0-indexed.

**multiply** \((A, B, n)\)

1. if \( n = 1 \) then return \((A[0]B[0])\)
2. \( A_L \leftarrow A[0..n/2 - 1], A_H \leftarrow A[n/2 .. n - 1] \)
3. \( B_L \leftarrow B[0..n/2 - 1], B_H \leftarrow B[n/2 .. n - 1] \)
4. \( C_L \leftarrow \text{multiply}(A_L, B_L, n/2) \)
5. \( C_H \leftarrow \text{multiply}(A_H, B_H, n/2) \)
6. \( C_M \leftarrow \text{multiply}(A_L + A_H, B_L + B_H, n/2) \)
7. \( C \leftarrow \text{array of } (2n - 1) \text{ 0's} \)
8. for \( i \leftarrow 0 \) to \( n - 2 \) do
9. \( C[i] \leftarrow C[i] + C_L[i] \)
10. \( C[i + n] \leftarrow C[i + n] + C_H[i] \)
11. \( C[i + n/2] \leftarrow C[i + n/2] + C_M[i] - C_L[i] - C_H[i] \)
12. return \( C \)
Outline

1. Divide-and-Conquer
2. Counting Inversions
3. Quicksort and Selection
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
4. Polynomial Multiplication
5. Other Classic Algorithms using Divide-and-Conquer
6. Solving Recurrences
7. Computing \( n \)-th Fibonacci Number
• Closest pair
• Convex hull
• Matrix multiplication
• FFT (Fast Fourier Transform): polynomial multiplication in $O(n \lg n)$ time
Closest Pair

**Input:** $n$ points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

**Output:** the pair of points that are closest
Closest Pair

**Input:** $n$ points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

**Output:** the pair of points that are closest
Closest Pair

**Input:** \( n \) points in plane: \((x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\)

**Output:** the pair of points that are closest

- **Trivial algorithm:** \( O(n^2) \) running time
Divide-and-Conquer Algorithm for Closest Pair

- **Divide**: Divide the points into two halves via a vertical line
Divide-and-Conquer Algorithm for Closest Pair

- **Divide**: Divide the points into two halves via a vertical line.
- **Conquer**: Solve two sub-instances recursively.
Divide-and-Conquer Algorithm for Closest Pair

- **Divide**: Divide the points into two halves via a vertical line
- **Conquer**: Solve two sub-instances recursively
- **Combine**: Check if there is a closer pair between left-half and right-half

![Diagram of points with a vertical line dividing them and arrows pointing to points within a certain distance δ.](image-url)
Divide-and-Conquer Algorithm for Closest Pair

Each box contains at most one pair

For each point, only need to consider \( O(1) \) boxes nearby

Time for combine = \( O(n) \) (many technicalities omitted)

Recurrence:
\[
T(n) = 2T(n/2) + O(n)
\]

Running time:
\( O(n \log n) \)
Each box contains at most one pair
Each box contains at most one pair.
For each point, only need to consider $O(1)$ boxes nearby.
Divide-and-Conquer Algorithm for Closest Pair

- Each box contains at most one pair
- For each point, only need to consider $O(1)$ boxes nearby
- time for combine = $O(n)$ (many technicalities omitted)
Divide-and-Conquer Algorithm for Closest Pair

- Each box contains at most one pair
- For each point, only need to consider $O(1)$ boxes nearby
- time for combine = $O(n)$ (many technicalities omitted)
- Recurrence: $T(n) = 2T(n/2) + O(n)$
Divide-and-Conquer Algorithm for Closest Pair

- Each box contains at most one pair
- For each point, only need to consider $O(1)$ boxes nearby
- Time for combine = $O(n)$ (many technicalities omitted)
- Recurrence: $T(n) = 2T(n/2) + O(n)$
- Running time: $O(n \lg n)$
$O(n \lg n)$-Time Algorithm for Convex Hull
$O(n \lg n)$-Time Algorithm for Convex Hull
$O(n \lg n)$-Time Algorithm for Convex Hull
$O(n \lg n)$-Time Algorithm for Convex Hull
$O(n \log n)$-Time Algorithm for Convex Hull
Strassen’s Algorithm for Matrix Multiplication

**Matrix Multiplication**

**Input:** two \( n \times n \) matrices \( A \) and \( B \)

**Output:** \( C = AB \)
Matrix Multiplication

Input: two $n \times n$ matrices $A$ and $B$

Output: $C = AB$

Naive Algorithm: $\text{matrix-multiplication}(A, B, n)$

1: for $i \leftarrow 1$ to $n$ do
2:   for $j \leftarrow 1$ to $n$ do
3:       $C[i, j] \leftarrow 0$
4:   for $k \leftarrow 1$ to $n$ do
5:       $C[i, j] \leftarrow C[i, j] + A[i, k] \times B[k, j]$
6: return $C$

running time = $O(n^3)$
Strassen’s Algorithm for Matrix Multiplication

Matrix Multiplication

Input: two $n \times n$ matrices $A$ and $B$
Output: $C = AB$

Naive Algorithm: matrix-multiplication($A, B, n$)

1: for $i \leftarrow 1$ to $n$ do
2:     for $j \leftarrow 1$ to $n$ do
3:         $C[i, j] \leftarrow 0$
4:     for $k \leftarrow 1$ to $n$ do
5:         $C[i, j] \leftarrow C[i, j] + A[i, k] \times B[k, j]$
6: return $C$

- running time $= O(n^3)$
Try to Use Divide-and-Conquer

\[
A = \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}, \quad \frac{n}{2}
\]

\[
B = \begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}, \quad \frac{n}{2}
\]

\[
C = \begin{pmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{pmatrix}
\]

\text{matrix\_multiplication}(A, B) \text{ recursively calls } \text{matrix\_multiplication}(A_{11}, B_{11}), \text{ matrix\_multiplication}(A_{12}, B_{21}), \ldots
Try to Use Divide-and-Conquer

\[
\begin{align*}
A &= \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix} \quad & B &= \begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\end{align*}
\]

\[
C = \begin{pmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{pmatrix}
\]

- \text{matrix\_multiplication}(A, B) recursively calls \text{matrix\_multiplication}(A_{11}, B_{11}), \text{matrix\_multiplication}(A_{12}, B_{21}), \ldots

- \text{Recurrence for running time: } T(n) = 8T(n/2) + O(n^2)

- \text{Recurrence for running time: } T(n) = O(n^3)
Strassen’s Algorithm

- \( T(n) = 8T(n/2) + O(n^2) \)
- Strassen’s Algorithm: improve the number of multiplications from 8 to 7!
- New recurrence: \( T(n) = 7T(n/2) + O(n^2) \)
Strassen’s Algorithm

- \( T(n) = 8T(n/2) + O(n^2) \)
- Strassen’s Algorithm: improve the number of multiplications from 8 to 7!
- New recurrence: \( T(n) = 7T(n/2) + O(n^2) \)
- Solving Recurrence \( T(n) = O(n^{\log_2 7}) = O(n^{2.808}) \)
Outline

1. Divide-and-Conquer
2. Counting Inversions
3. Quicksort and Selection
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
4. Polynomial Multiplication
5. Other Classic Algorithms using Divide-and-Conquer
6. Solving Recurrences
7. Computing $n$-th Fibonacci Number
Methods for Solving Recurrences

- The recursion-tree method
- The master theorem
Recursion-Tree Method

\[ T(n) = 2T(n/2) + O(n) \]
Recursion-Tree Method

- \( T(n) = 2T(n/2) + O(n) \)
Recursion-Tree Method

- $T(n) = 2T(n/2) + O(n)$

Each level takes running time $O(n)$
Recursion-Tree Method

- \( T(n) = 2T(n/2) + O(n) \)

Each level takes running time \( O(n) \)

There are \( O(\lg n) \) levels
Recursion-Tree Method

- \( T(n) = 2T(n/2) + O(n) \)

Each level takes running time \( O(n) \)

There are \( O(\lg n) \) levels

Running time = \( O(n \lg n) \)
Recursion-Tree Method

\[ T(n) = 3T(n/2) + O(n) \]
Recursion-Tree Method

\[ T(n) = 3T(n/2) + O(n) \]
Recursion-Tree Method

\[ T(n) = 3T(n/2) + O(n) \]
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n) \)
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n) \)
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n) \)

- Total running time at level \( i \)?
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n) \)

![Diagram of recursion tree]

- Total running time at level \( i \)? \( \frac{n}{2^i} \times 3^i = \left(\frac{3}{2}\right)^i n \)
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n) \)

![Recursion Tree Diagram]

- Total running time at level \( i \)? \( \frac{n}{2^i} \times 3^i = \left(\frac{3}{2}\right)^i n \)
- Index of last level?
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n) \)

\[
T(n) = 3T(n/2) + O(n)
\]

\[
\begin{align*}
T(n) &= 3T(n/2) + O(n) \\
     &\vdots \\
\end{align*}
\]

- Total running time at level \( i \)? \( \frac{n}{2^i} \times 3^i = \left(\frac{3}{2}\right)^i n \)
- Index of last level? \( \lg_2 n \)
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n) \)

![Recursion Tree Diagram]

- Total running time at level \( i \)? \( \frac{n}{2^i} \times 3^i = \left(\frac{3}{2}\right)^i n \)
- Index of last level? \( \lg_2 n \)
- Total running time?
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n) \)

Total running time at level \( i \)? \( \frac{n}{2^i} \times 3^i = \left( \frac{3}{2} \right)^i n \)

Index of last level? \( \lg_2 n \)

Total running time?

\[
\sum_{i=0}^{\lg_2 n} \left( \frac{3}{2} \right)^i n = O \left( n \left( \frac{3}{2} \right)^{\lg_2 n} \right) = O(3^{\lg_2 n}) = O(n^{\lg_2 3}).
\]
Recursion-Tree Method

\[ T(n) = 3T\left(\frac{n}{2}\right) + O(n^2) \]
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n^2) \)
Recursion-Tree Method

\[ T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n^2) \]
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n^2) \)
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n^2) \)
**Recursion-Tree Method**

- \( T(n) = 3T(n/2) + O(n^2) \)

![Recursion Tree Diagram]

- Total running time at level \( i \)?

\[ \sum_{i=0}^{\log_2 n} \left( \frac{3}{4} \right)^i n^2 = O(n^2) \]
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n^2) \)

- Total running time at level \( i \)? \( \left( \frac{n}{2^i} \right)^2 \times 3^i = \left( \frac{3}{4} \right)^i n^2 \)
Recursion-Tree Method

- $T(n) = 3T(n/2) + O(n^2)$

$\begin{array}{c}
\text{Level 0: } n^2 \\
\text{Level 1: } (n/2)^2, (n/2)^2, (n/2)^2 \\
\text{Level 2: } (n/4)^2, (n/4)^2, (n/4)^2, (n/4)^2, (n/4)^2 \\
\text{Level 3: } (n/8)^2, (n/8)^2, (n/8)^2, (n/8)^2, (n/8)^2, (n/8)^2, (n/8)^2 \\
\end{array}$

- Total running time at level $i$: $(\frac{n}{2^i})^2 \times 3^i = (\frac{3}{4})^i n^2$
- Index of last level?

$\sum_{i=0}^{\log_2 n} (\frac{3}{4})^i n^2 = O(n^2)$
**Recursion-Tree Method**

- \( T(n) = 3T(n/2) + O(n^2) \)

![Recursion Tree Diagram]

- Total running time at level \( i \)? \( (\frac{n}{2^i})^2 \times 3^i = \left(\frac{3}{4}\right)^i n^2 \)
- Index of last level? \( 1g_2 n \)
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n^2) \)

- Total running time at level \( i \)? \( \left( \frac{n}{2^i} \right)^2 \times 3^i = \left( \frac{3}{4} \right)^i n^2 \)

- Index of last level? \( \lg_2 n \)

- Total running time?
Recursion-Tree Method

- \[ T(n) = 3T\left(\frac{n}{2}\right) + O(n^2) \]

- Total running time at level \( i \)? \( \left(\frac{n}{2^i}\right)^2 \times 3^i = \left(\frac{3}{4}\right)^i n^2 \)
- Index of last level? \( \lg_2 n \)
- Total running time?

\[
\sum_{i=0}^{\lg_2 n} \left(\frac{3}{4}\right)^i n^2 =
\]
Recursion-Tree Method

- \( T(n) = 3T(n/2) + O(n^2) \)

- Total running time at level \( i \)? \( (\frac{n}{2^i})^2 \times 3^i = (\frac{3}{4})^i n^2 \)
- Index of last level? \( \log_2 n \)
- Total running time?

\[
\sum_{i=0}^{\log_2 n} \left(\frac{3}{4}\right)^i n^2 = O(n^2).
\]
Master Theorem

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td></td>
<td></td>
<td></td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n)$</td>
<td></td>
<td></td>
<td></td>
<td>$O(n^{\lg_2 3})$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n^2)$</td>
<td></td>
<td></td>
<td></td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**Theorem** \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,
Master Theorem

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T(n) = 2T(n/2) + O(n) )</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>( O(n \log n) )</td>
</tr>
<tr>
<td>( T(n) = 3T(n/2) + O(n) )</td>
<td></td>
<td></td>
<td></td>
<td>( O(n^{\log_2 3}) )</td>
</tr>
<tr>
<td>( T(n) = 3T(n/2) + O(n^2) )</td>
<td></td>
<td></td>
<td></td>
<td>( O(n^2) )</td>
</tr>
</tbody>
</table>

**Theorem** \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,
**Master Theorem**

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n)$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>$O(n^{\lg_2 3})$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n^2)$</td>
<td></td>
<td></td>
<td></td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**Theorem**  

$T(n) = aT(n/b) + O(n^c)$, where $a \geq 1$, $b > 1$, $c \geq 0$ are constants. Then,
Master Theorem

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n)$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>$O(n^{\lg_2 3})$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n^2)$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**Theorem** $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1$, $b > 1$, $c \geq 0$ are constants. Then,
**Master Theorem**

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n)$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>$O(n^{\lg_2 3})$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n^2)$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**Theorem**  

$T(n) = aT(n/b) + O(n^c)$, where $a \geq 1$, $b > 1$, $c \geq 0$ are constants. Then,

$$
T(n) = \begin{cases} 
\text{if } c < \lg_b a \\
\text{if } c = \lg_b a \\
\text{if } c > \lg_b a 
\end{cases}
$$
Master Theorem

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n)$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>$O(n^{\lg_2 3})$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n^2)$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**Theorem** $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1$, $b > 1$, $c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} 
?? & \text{if } c < \lg_b a \\
?? & \text{if } c = \lg_b a \\
?? & \text{if } c > \lg_b a
\end{cases}$$
Master Theorem

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n)$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>$O(n^{\lg_2 3})$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n^2)$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**Theorem** \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c) & \text{if } c = \lg_b a \\
O(n^2) & \text{if } c > \lg_b a
\end{cases}
\]
Master Theorem

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n)$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>$O(n^{\lg_2 3})$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n^2)$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

Theorem \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
\text{??} & \text{if } c = \lg_b a \\
\text{??} & \text{if } c > \lg_b a 
\end{cases}
\]
Master Theorem

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n)$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>$O(n^{\lg_2 3})$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n^2)$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**Theorem** \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a 
\end{cases}
\]
### Master Theorem

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$O(n \lg n)$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n)$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>$O(n^{\lg_2 3})$</td>
</tr>
<tr>
<td>$T(n) = 3T(n/2) + O(n^2)$</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

**Theorem** $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1$, $b > 1$, $c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} 
O(n^{\log_b a}) & \text{if } c < \log_b a \\
? & \text{if } c = \log_b a \\
O(n^c) & \text{if } c > \log_b a 
\end{cases}$$
# Master Theorem

Theorem \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a 
\end{cases}
\]

## Recurrences

<table>
<thead>
<tr>
<th>Recurrences</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T(n) = 2T(n/2) + O(n) )</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>( O(n \lg n) )</td>
</tr>
<tr>
<td>( T(n) = 3T(n/2) + O(n) )</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>( O(n^{\lg_2 3}) )</td>
</tr>
<tr>
<td>( T(n) = 3T(n/2) + O(n^2) )</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>( O(n^2) )</td>
</tr>
</tbody>
</table>
**Theorem** \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a 
\end{cases}
\]

- **Ex:** \( T(n) = 4T(n/2) + O(n^2) \). Which Case?
**Theorem** \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a 
\end{cases}
\]

- Ex: \( T(n) = 4T(n/2) + O(n^2) \). **Case 2.**
Theorem \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^\lfloor \lg_b a \rfloor) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a
\end{cases}
\]

- Ex: \( T(n) = 4T(n/2) + O(n^2) \). Case 2. \( T(n) = O(n^2 \lg n) \)
Theorem \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a
\end{cases}
\]

- Ex: \( T(n) = 4T(n/2) + O(n^2) \). Case 2. \( T(n) = O(n^2 \lg n) \)
- Ex: \( T(n) = 3T(n/2) + O(n) \). Which Case?
**Theorem** \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a 
\end{cases}
\]

- Ex: \( T(n) = 4T(n/2) + O(n^2) \). Case 2. \( T(n) = O(n^2 \lg n) \)
- Ex: \( T(n) = 3T(n/2) + O(n) \). Case 1.
**Theorem** \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a 
\end{cases}
\]

- Ex: \( T(n) = 4T(n/2) + O(n^2) \). Case 2. \( T(n) = O(n^2 \lg n) \)
- Ex: \( T(n) = 3T(n/2) + O(n) \). Case 1. \( T(n) = O(n^{\lg_2 3}) \)
Theorem \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a 
\end{cases}
\]

- Ex: \( T(n) = 4T(n/2) + O(n^2) \). Case 2. \( T(n) = O(n^2 \lg n) \)
- Ex: \( T(n) = 3T(n/2) + O(n) \). Case 1. \( T(n) = O(n^{\lg_2 3}) \)
- Ex: \( T(n) = T(n/2) + O(1) \). Which Case?
Theorem \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a 
\end{cases}
\]

- Ex: \( T(n) = 4T(n/2) + O(n^2) \). Case 2. \( T(n) = O(n^2 \lg n) \)
- Ex: \( T(n) = 3T(n/2) + O(n) \). Case 1. \( T(n) = O(n^{\lg_2 3}) \)
- Ex: \( T(n) = T(n/2) + O(1) \). Case 2.
**Theorem** \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a 
\end{cases}
\]

- **Ex:** \( T(n) = 4T(n/2) + O(n^2) \). Case 2. \( T(n) = O(n^2 \lg n) \)
- **Ex:** \( T(n) = 3T(n/2) + O(n) \). Case 1. \( T(n) = O(n^{\lg_2 3}) \)
- **Ex:** \( T(n) = T(n/2) + O(1) \). Case 2. \( T(n) = O(\lg n) \)
**Theorem** \[ T(n) = aT(n/b) + O(n^c), \] where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^\lfloor \lg_b a \rfloor) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a 
\end{cases}
\]

- **Ex:** \( T(n) = 4T(n/2) + O(n^2) \). Case 2. \( T(n) = O(n^2 \lg n) \)
- **Ex:** \( T(n) = 3T(n/2) + O(n) \). Case 1. \( T(n) = O(n^{\lg_2 3}) \)
- **Ex:** \( T(n) = T(n/2) + O(1) \). Case 2. \( T(n) = O(\lg n) \)
- **Ex:** \( T(n) = 2T(n/2) + O(n^2) \). Which Case?
Theorem \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases}
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \log n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a
\end{cases}
\]

- Ex: \( T(n) = 4T(n/2) + O(n^2) \). Case 2. \( T(n) = O(n^2 \log n) \)
- Ex: \( T(n) = 3T(n/2) + O(n) \). Case 1. \( T(n) = O(n^{\lg_2 3}) \)
- Ex: \( T(n) = T(n/2) + O(1) \). Case 2. \( T(n) = O(\log n) \)
- Ex: \( T(n) = 2T(n/2) + O(n^2) \). Case 3.
**Theorem**  \( T(n) = aT(n/b) + O(n^c) \), where \( a \geq 1, b > 1, c \geq 0 \) are constants. Then,

\[
T(n) = \begin{cases} 
O(n^{\lg_b a}) & \text{if } c < \lg_b a \\
O(n^c \lg n) & \text{if } c = \lg_b a \\
O(n^c) & \text{if } c > \lg_b a 
\end{cases}
\]

- **Ex:** \( T(n) = 4T(n/2) + O(n^2) \). Case 2. \( T(n) = O(n^2 \lg n) \)
- **Ex:** \( T(n) = 3T(n/2) + O(n) \). Case 1. \( T(n) = O(n^{\lg_2 3}) \)
- **Ex:** \( T(n) = T(n/2) + O(1) \). Case 2. \( T(n) = O(\lg n) \)
- **Ex:** \( T(n) = 2T(n/2) + O(n^2) \). Case 3. \( T(n) = O(n^2) \)
Proof of Master Theorem Using Recursion Tree

\[ T(n) = aT\left(\frac{n}{b}\right) + O(n^c) \]

1 node

\[ n^c \]

\[ a \text{ nodes} \]

\[ (n/b)^c \]

\[ a^2 \text{ nodes} \]

\[ (n/b^2)^c \]

\[ (n/b^2)^c \]

\[ (n/b^2)^c \]

\[ (n/b^2)^c \]

\[ a^3 \text{ nodes} \]

\[ \left(\frac{n}{b^3}\right)^c \]

\[ \left(\frac{n}{b^3}\right)^c \]

\[ \left(\frac{n}{b^3}\right)^c \]

\[ \left(\frac{n}{b^3}\right)^c \]

\[ \left(\frac{n}{b^3}\right)^c \]

\[ \left(\frac{n}{b^3}\right)^c \]

\[ \left(\frac{n}{b^3}\right)^c \]
Proof of Master Theorem Using Recursion Tree

\[ T(n) = aT(n/b) + O(n^c) \]

- 1 node
- \( n^c \)
- \( n^c \)
- \( \frac{a}{b^c} n^c \)
- \( a \) nodes
- \( (n/b)^c \)
- \( (n/b)^c \)
- \( \frac{a}{b^c} n^c \)
- \( a^2 \) nodes
- \( (n/b^2)^c \)
- \( (n/b^2)^c \)
- \( \frac{a}{b^c} n^c \)
- \( (n/b^2)^c \)
- \( (n/b^2)^c \)
- \( \frac{a}{b^c} n^c \)
- \( a^3 \) nodes
- \( \frac{n}{b^3}^c \)
- \( \frac{n}{b^3}^c \)
- \( \frac{n}{b^3}^c \)
- \( \frac{n}{b^3}^c \)
- \( \frac{n}{b^3}^c \)
- \( \frac{n}{b^3}^c \)
- \( \frac{n}{b^3}^c \)
- \( \frac{n}{b^3}^c \)
- \( \frac{n}{b^3}^c \)
Proof of Master Theorem Using Recursion Tree

\[ T(n) = aT(n/b) + O(n^c) \]

- 1 node
- \( n^c \)
- \( a \) nodes
- \( (n/b)^c \)
- \( (n/b)^c \)
- \( \frac{a}{b^c} n^c \)
- \( a^2 \) nodes
- \( (n/b^2)^c \)
- \( (n/b^2)^c \)
- \( (n/b^2)^c \)
- \( \left(\frac{a}{b^c}\right)^2 n^c \)
- \( a^3 \) nodes
- \( \left(\frac{n}{b^3}\right)^c \)
- \( \left(\frac{n}{b^3}\right)^c \)
- \( \left(\frac{n}{b^3}\right)^c \)
- \( \left(\frac{n}{b^3}\right)^c \)
- \( \left(\frac{n}{b^3}\right)^c \)
- \( \left(\frac{n}{b^3}\right)^c \)
- \( \left(\frac{n}{b^3}\right)^c \)
- \( \left(\frac{n}{b^3}\right)^c \)
- \( \left(\frac{n}{b^3}\right)^c \)
- \( \left(\frac{n}{b^3}\right)^c \)

- \( c < \log_b a \): bottom-level dominates: \( \left(\frac{a}{b^c}\right)^{\log_b n} n^c = n^{\log_b a} \)
Proof of Master Theorem Using Recursion Tree

\[ T(n) = aT\left(\frac{n}{b}\right) + O(n^c) \]

- **1 node**
  - \( n^c \)

- **1 node**
  - \( n^c \)

- **a nodes**
  - \( \left(\frac{n}{b}\right)^c \)
  - \( \left(\frac{n}{b}\right)^c \)

- **a^2 nodes**
  - \( \left(\frac{n}{b^2}\right)^c \)
  - \( \left(\frac{n}{b^2}\right)^c \)

- **a^3 nodes**
  - \( \left(\frac{n}{b^3}\right)^c \)
  - \( \left(\frac{n}{b^3}\right)^c \)
  - \( \left(\frac{n}{b^3}\right)^c \)
  - \( \left(\frac{n}{b^3}\right)^c \)

\[ c < \lg_b a : \text{bottom-level dominates: } \left(\frac{a}{b^c}\right)^{\lg_b n} n^c = n^{\lg_b a} \]

\[ c = \lg_b a : \text{all levels have same time: } n^c \lg_b n = O(n^c \lg n) \]
Proof of Master Theorem Using Recursion Tree

\[ T(n) = aT\left(\frac{n}{b}\right) + O(n^c) \]

- **\( a \) nodes**: 
  - \( (n/b)^c \)
  - \( (n/b)^c \)
  - \( (n/b)^c \)
  - \( (n/b)^c \)
  - \( \left(\frac{n}{b^3}\right)^c \)
  - \( \left(\frac{n}{b^3}\right)^c \)

- **\( a^2 \) nodes**: 
  - \( (n/b^2)^c \)
  - \( (n/b^2)^c \)
  - \( (n/b^2)^c \)
  - \( (n/b^2)^c \)
  - \( \left(\frac{n}{b^3}\right)^c \)
  - \( \left(\frac{n}{b^3}\right)^c \)

- **\( a^3 \) nodes**: 
  - \( \left(\frac{n}{b^3}\right)^c \)
  - \( \left(\frac{n}{b^3}\right)^c \)
  - \( \left(\frac{n}{b^3}\right)^c \)
  - \( \left(\frac{n}{b^3}\right)^c \)
  - \( \left(\frac{n}{b^3}\right)^c \)
  - \( \left(\frac{n}{b^3}\right)^c \)

- **\( c < \log_b a \)**: bottom-level dominates: \( \left(\frac{a}{b^c}\right)^{\log_b n} n^c = n^{\log_b a} \)

- **\( c = \log_b a \)**: all levels have same time: \( n^c \log_b n = O(n^c \log n) \)

- **\( c > \log_b a \)**: top-level dominates: \( O(n^c) \)
Outline

1. Divide-and-Conquer
2. Counting Inversions
3. Quicksort and Selection
   - Quicksort
   - Lower Bound for Comparison-Based Sorting Algorithms
   - Selection Problem
4. Polynomial Multiplication
5. Other Classic Algorithms using Divide-and-Conquer
6. Solving Recurrences
7. Computing $n$-th Fibonacci Number
Fibonacci Numbers

- $F_0 = 0, F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}, \forall n \geq 2$
- Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots

\textbf{n-th Fibonacci Number}

\begin{itemize}
  \item \textbf{Input:} integer $n > 0$
  \item \textbf{Output:} $F_n$
\end{itemize}
Computing $F_n$ : Stupid Divide-and-Conquer Algorithm

**Fib($n$)**

1: if $n = 0$ return 0  
2: if $n = 1$ return 1  
3: return $\text{Fib}(n - 1) + \text{Fib}(n - 2)$

**Q:** Is the running time of the algorithm polynomial or exponential in $n$?
Computing $F_n$: Stupid Divide-and-Conquer Algorithm

**Fib($n$)**

1. if $n = 0$ return 0
2. if $n = 1$ return 1
3. return $\text{Fib}(n - 1) + \text{Fib}(n - 2)$

**Q:** Is the running time of the algorithm polynomial or exponential in $n$?

**A:** Exponential
Computing $F_n$ : Stupid Divide-and-Conquer Algorithm

<table>
<thead>
<tr>
<th>Fib(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: if $n = 0$ return 0</td>
</tr>
<tr>
<td>2: if $n = 1$ return 1</td>
</tr>
<tr>
<td>3: return Fib($n - 1$) + Fib($n - 2$)</td>
</tr>
</tbody>
</table>

Q: Is the running time of the algorithm polynomial or exponential in $n$?

A: Exponential

Running time is at least $\Omega(F_n)$
Computing $F_n$: Stupid Divide-and-Conquer Algorithm

**Fib($n$)**

1: if $n = 0$ return 0
2: if $n = 1$ return 1
3: return Fib($n - 1$) + Fib($n - 2$)

**Q:** Is the running time of the algorithm polynomial or exponential in $n$?

**A:** Exponential

- Running time is at least $\Omega(F_n)$
- $F_n$ is exponential in $n$
Computing \( F_n \): Reasonable Algorithm

\[
\text{Fib}(n)
\]

1: \( F[0] \leftarrow 0 \)
2: \( F[1] \leftarrow 1 \)
3: \( \text{for } i \leftarrow 2 \text{ to } n \text{ do} \)
4: \( F[i] \leftarrow F[i - 1] + F[i - 2] \)
5: \( \text{return } F[n] \)

- Dynamic Programming

Running time = ?
Computing $F_n$: Reasonable Algorithm

Fib($n$)

1: $F[0] \leftarrow 0$
2: $F[1] \leftarrow 1$
3: for $i \leftarrow 2$ to $n$ do
4: \hspace{1em} $F[i] \leftarrow F[i - 1] + F[i - 2]$
5: \hspace{1em} return $F[n]$

- Dynamic Programming
- Running time $= ?$
Computing $F_n$: Reasonable Algorithm

Fib($n$)

1: $F[0] \leftarrow 0$
2: $F[1] \leftarrow 1$
3: for $i \leftarrow 2$ to $n$ do
4: \hspace{1em} $F[i] \leftarrow F[i - 1] + F[i - 2]$
5: return $F[n]$

- Dynamic Programming
- Running time = $O(n)$
Computing $F_n$: Even Better Algorithm

\[
\begin{pmatrix}
F_n \\
F_{n-1}
\end{pmatrix}
= 
\begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix}
\begin{pmatrix}
F_{n-1} \\
F_{n-2}
\end{pmatrix}
\]

\[
\begin{pmatrix}
F_n \\
F_{n-1}
\end{pmatrix}
= 
\begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix}^2
\begin{pmatrix}
F_{n-2} \\
F_{n-3}
\end{pmatrix}
\]

\[\ldots\]

\[
\begin{pmatrix}
F_n \\
F_{n-1}
\end{pmatrix}
= 
\begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix}^{n-1}
\begin{pmatrix}
F_1 \\
F_0
\end{pmatrix}
\]
**power(n)**

1: if $n = 0$ then return $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
2: $R \leftarrow \text{power}([n/2])$
3: $R \leftarrow R \times R$
4: if $n$ is odd then $R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
5: return $R$

**Fib(n)**

1: if $n = 0$ then return 0
2: $M \leftarrow \text{power}(n - 1)$
3: return $M[1][1]$
**power(n)**

1: if $n = 0$ then return $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
2: $R \leftarrow \text{power}([n/2])$
3: $R \leftarrow R \times R$
4: if $n$ is odd then $R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
5: return $R$

**Fib(n)**

1: if $n = 0$ then return 0
2: $M \leftarrow \text{power}(n - 1)$
3: return $M[1][1]$

- Recurrence for running time?
**power**($n$)

1: if $n = 0$ then return $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

2: $R \leftarrow \text{power}([n/2])$

3: $R \leftarrow R \times R$

4: if $n$ is odd then $R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$

5: return $R$

**Fib**($n$)

1: if $n = 0$ then return 0

2: $M \leftarrow \text{power}(n - 1)$

3: return $M[1][1]$

- Recurrence for running time? $T(n) = T(n/2) + O(1)$
**power(n)**

1. if $n = 0$ then return $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
2. $R \leftarrow \text{power}([n/2])$
3. $R \leftarrow R \times R$
4. if $n$ is odd then $R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
5. return $R$

**Fib(n)**

1. if $n = 0$ then return 0
2. $M \leftarrow \text{power}(n - 1)$
3. return $M[1][1]$

- Recurrence for running time? $T(n) = T(n/2) + O(1)$
- $T(n) = O(\lg n)$
Running time $= O(\lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

A: $\Theta(n)$

We cannot add (or multiply) two integers of $\Theta(n)$ bits in $O(1)$ time.

Even printing $F(n)$ requires time much larger than $O(\lg n)$.

Fixing the Problem

To compute $F(n)$, we need $O(\lg n)$ basic arithmetic operations on integers.
Running time $= O(lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

Fixing the Problem
To compute $F(n)$, we need $O(lg n)$ basic arithmetic operations on integers.
Running time = $O(\lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

A: $\Theta(n)$
Running time $= O(\lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

A: $\Theta(n)$

- We cannot add (or multiply) two integers of $\Theta(n)$ bits in $O(1)$ time.
Running time $= O(lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

A: $\Theta(n)$

- We cannot add (or multiply) two integers of $\Theta(n)$ bits in $O(1)$ time.
- Even printing $F(n)$ requires time much larger than $O(lg n)$. 

Fixing the Problem

To compute $F(n)$, we need $O(lg n)$ basic arithmetic operations on integers.
**Running time = $O(\lg n)$: We Cheated!**

**Q:** How many bits do we need to represent $F(n)$?

**A:** $\Theta(n)$

- We can not add (or multiply) two integers of $\Theta(n)$ bits in $O(1)$ time.
- Even printing $F(n)$ requires time much larger than $O(\lg n)$.

**Fixing the Problem**

To compute $F_n$, we need $O(\lg n)$ basic arithmetic operations on integers.
Summary: Divide-and-Conquer

- **Divide**: Divide instance into many smaller instances
- **Conquer**: Solve each of smaller instances recursively and separately
- **Combine**: Combine solutions to small instances to obtain a solution for the original big instance
Summary: Divide-and-Conquer

- **Divide**: Divide instance into many smaller instances
- **Conquer**: Solve each of smaller instances recursively and separately
- **Combine**: Combine solutions to small instances to obtain a solution for the original big instance

- Write down recurrence for running time
- Solve recurrence using master theorem
Summary: Divide-and-Conquer

- Merge sort, quicksort, count-inversions, closest pair, \cdots:
  \[ T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \lg n) \]
Summary: Divide-and-Conquer

- Merge sort, quicksort, count-inversions, closest pair, ⋯:
  \[ T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n) \]

- Integer Multiplication:
  \[ T(n) = 3T(n/2) + O(n) \Rightarrow T(n) = O(n^{\log_2 3}) \]
Summary: Divide-and-Conquer

- **Merge sort, quicksort, count-inversions, closest pair, · · ·:**
  \[ T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n) \]

- **Integer Multiplication:**
  \[ T(n) = 3T(n/2) + O(n) \Rightarrow T(n) = O(n^{\log_2 3}) \]

- **Matrix Multiplication:**
  \[ T(n) = 7T(n/2) + O(n^2) \Rightarrow T(n) = O(n^{\log_2 7}) \]
Summary: Divide-and-Conquer

- Merge sort, quicksort, count-inversions, closest pair, ...:
  \[ T(n) = 2T\left(\frac{n}{2}\right) + O(n) \Rightarrow T(n) = O(n \lg n) \]

- Integer Multiplication:
  \[ T(n) = 3T\left(\frac{n}{2}\right) + O(n) \Rightarrow T(n) = O(n^{\lg_2 3}) \]

- Matrix Multiplication:
  \[ T(n) = 7T\left(\frac{n}{2}\right) + O(n^2) \Rightarrow T(n) = O(n^{\lg_2 7}) \]

- Usually, designing better algorithm for “combine” step is key to improve running time